JavaScript Callbacks

1. Introduction to Callbacks

In JavaScript, a callback is a function passed as an argument to another function, which is then executed at a later time. Callbacks enable JavaScript to perform tasks asynchronously, such as fetching data from a server or waiting for user interactions, without blocking the rest of the program.

Why are callbacks useful?
- JavaScript is single-threaded and can't run multiple tasks simultaneously.
- Callbacks enable non-blocking behavior, allowing the program to handle other operations while waiting for long tasks to complete.
- Common use cases include API requests, event listeners, and timer functions.

2. Synchronous vs Asynchronous JavaScript

Problem:
In synchronous JavaScript, code runs one line at a time. Long-running tasks block the execution of subsequent lines.

Example of Synchronous Code:

```
console.log("1");
console.log("2");
console.log("3");
```

All three log statements run in order.

Solution:
Asynchronous code allows certain operations to be started and completed later, without halting the rest of the program.

Example of Asynchronous Code:

```
console.log("1");
setTimeout(() => console.log("2"), 2000);
console.log("3");
```

Output: 1, then 3, and after 2 seconds, 2.

Future Guide:
Use async operations (like setTimeout, fetch, etc.) with callbacks to avoid freezing the app while waiting.

3. JavaScript Is Single-Threaded

Problem:
JavaScript executes in a single thread, meaning it can only do one thing at a time. Long tasks block everything else.

Solution:
Use asynchronous patterns (callbacks, promises, async/await) to let JavaScript handle multiple tasks efficiently.

Future Guide:
Use event loop concepts to understand how JavaScript queues and processes async code behind the scenes.

4. Callback Syntax and Examples

4.1 Basic Function Callback

Problem:
You want to run one function after another completes.

Solution:
Pass one function as a callback to another.

```
function greetUser(name, callback) {
console.log("Hello " + name);
callback();
}

function sayGoodbye() {
 console.log("Goodbye!");
}

greetUser("Alice", sayGoodbye);
```

Future Guide:
Use callbacks when order of operations matters.

4.2 Event Listener Callback

Problem:
You need to run code only when a user clicks a button.

Solution:
Use callbacks with `addEventListener`.

```
document.getElementById("new-deck").addEventListener("click",
function() {

fetch("https://apis.scrimba.com/deckofcards/api/deck/new/shuffle/")
```

```
.then(res => res.json())
.then(data => console.log(data));
});
```

Future Guide:
Separate named callback functions are cleaner and reusable.

4.3 setTimeout Callback

Problem:
You want to delay code execution.

Solution:
Use `setTimeout` with a callback function.

```
function callback() {
console.log("I finally ran!");
}

setTimeout(callback, 2000);
```

Future Guide:
Timers help simulate asynchronous behavior.

4.4 Array.filter Callback

Problem:
You want to filter an array based on a condition.

Solution:
Use the built-in `.filter()` method with a callback.

```
const people = [
{ name: "Jack", hasPet: true },
{ name: "Jill", hasPet: false },
{ name: "Alice", hasPet: true },
{ name: "Bob", hasPet: false },
];

const peopleWithPets = people.filter(person => person.hasPet);
console.log(peopleWithPets);
```

Future Guide:
Most array methods accept callbacks (map, reduce, forEach, etc).

4.5 Custom filterArray Function

Problem:
You want to understand how .filter() works under the hood.

Solution:
Write your own `filterArray()` function using a callback.

```
function filterArray(array, callback) {
const result = [];
for (let item of array) {
if (callback(item)) {
result.push(item);
}
}
return result;
}

const pets = filterArray(people, person => person.hasPet);
console.log(pets);
```

Future Guide:
This helps in understanding how array functions internally leverage callbacks.

5. Real-World Problem Solved Using Callbacks

Problem:
You're building a web app that loads user data from an API, processes it (e.g., filters users who are active), and then displays the result. Each step depends on the previous one completing successfully.

Solution:
Use nested callbacks to handle asynchronous flow of loading, processing, and displaying data.

```
function fetchUserData(callback) {
setTimeout(() => {
const users = [
{ name: "Alice", isActive: true },
{ name: "Bob", isActive: false },
{ name: "Carol", isActive: true }
];
console.log("Fetched users");
callback(users);
}, 1000);
}

function filterActiveUsers(users, callback) {
setTimeout(() => {
const activeUsers = users.filter(user => user.isActive);
console.log("Filtered active users");
callback(activeUsers);
}, 1000);
```

```
}

function displayUsers(users) {
 console.log("Displaying users:");
users.forEach(user => console.log(user.name));
}

fetchUserData(function(users) {
filterActiveUsers(users, function(activeUsers) {
 displayUsers(activeUsers);
 });
});
```

Explanation:
- `fetchUserData`: Simulates an asynchronous fetch operation.
- `filterActiveUsers`: Processes the data with another async simulation.
- `displayUsers`: Final step to show data.

Callbacks help ensure that each step only runs after the previous one has finished. This is essential in asynchronous programming where operations may complete at different times.

Future Guide:
This approach is called 'callback chaining'. It works but can get messy (a situation called 'callback hell'). Eventually, Promises and async/await are better tools to simplify this pattern.